Challenges to a standard monitoring interface

Stéphane Eranian
HPLabs
eranian@hpl.hp.com
HPM workshop, MICRO-39, Dec 2006 Orlando, FL

For the last few years, we have been working on designing and implementing a standard monitoring interface for Linux to access the hardware performance counters of modern processors. In this presentation, we describe some of the challenges we encountered during this work as well as some we are facing now. Our hope is to spark interesting discussions and connect with peers to try and solve them.

The goal of a monitoring interface is to provide user access to a piece of processor hardware called the Performance Monitoring Unit (PMU). Monitoring tools use this interface to collect counts and profiles on a per-thread or system-wide basis, i.e. across all threads and all processors. Accessing the PMU requires, for all architectures, executing at the most privileged level, i.e., in the kernel. As such, a monitoring interface is a new kernel interface. Designing this interface requires solving challenges at all three levels: hardware, operating system, and user.

One of the key values of a standard monitoring interface is to provide a uniform set of features and behaviors across all platforms. By definition, the PMU is very specific to the implementation of each processor. As such the interface must manage a wide variety of hardware features and behaviors without being forced into taking the lowest common denominator approach.

We believe that the lack of true PMU architecture in some processor families makes it hard to design and implement a monitoring interface guaranteed work with future processors. An architecture defines framework within which the PMU can evolve. It also defines the basic behaviors, such as how to start and stop, what happens on counter overflow, how to detect which counter overflowed, power-on values, low-power behavior, minimal set of events. Without such framework, for each new implementation, we may have to add more model-specific code and create new internal interfaces to mask differences. For instance, with the Intel Pentium III, Pentium 4, Core 2 Duo processors, we have three different ways of detecting which counters overflowed. In contrast, there is a single routine for all Itanium processors, simply because overflow detection is architected. Model specific code is reduced to a description of where the registers are and it could be delivered as a kernel module, thereby minimizing the dependency on Linux distribution release cycles for enabling new hardware.

A standard monitoring interface cannot assume that all users know exactly what they are doing. There are security but also correctness issues. For instance, when multi-threading is enabled on the Pentium 4, the eighteen counters are shared between the two threads. What happens when the operating system schedules two processes on logical CPUs sharing the same core? Similarly, Precise-Event-Based Sampling (PEBS) does not work when multi-threading is enabled. On the AMD Opteron, some counters are global, as such only one session can access them at any one time. On Itanium 2, the code debug registers may be used to restrict monitoring to code range. What happens if a system-wide monitoring session uses this feature at the same time a thread is debugged using the same registers?

Sharing the PMU resource between multiple conflicting users is still a challenge we are facing. For instance, it is not possible to have a system-wide session running in parallel with a per-thread session if both assume they own the entire PMU. Existing interfaces avoid such conflict by enforcing mutual exclusion between the two types of session. This restriction must be lifted fairly soon as some applications are using the PMU systematically. Today, certain PMU limitations make this difficult to support, such as the dependencies between PMU registers, a common start/stop control. There are also some more generic issues such as how to design a generic PMU register allocator, do we need to use priorities to access PMU registers. There are also consequences for tools which must be prepared to fail and use alternate PMU registers for a given event.

A virtual machine monitor (VMM) must virtualize the PMU to allow concurrent accesses by tools running on guest operating systems, something missing from the Xen virtual machine today. There are issues associated with the cost of accessing the PMU on domain switches. With para-virtualization, a guest OS does not run at privilege level 0 anymore. Consequently, on some architectures, counters may not be able to measure a guest OS. That also raises the question as to what are user tools supposed to measure: the guest OS, the VMM or both. Hardware support for virtualization does help to some extent but it does not solve everything. In a virtual environment, a system-wide session measures across the VMM and all guests OSes therefore the PMU accesses would also have to be coordinated.

The PMU is still fairly difficult to understand and exploit for non-experts. The documentation is too often unclear. Figuring out how to collect some basic key metrics out of a list of several hundred events can be challenging. We believe each PMU should make it clear how to collect the following metrics: floating-point operations per second (FLOPS), cycles per instruction (CPI), stalls, code and data cache/TLB misses, system/memory bus bandwidth utilization, software prefetch effectiveness. For instance, it is hard to compute FLOPS on an AMD Opteron processor. Tools should become smarter at interpreting PMU data and presenting it in a meaningful manner to users. Compilers needs to better exploit PMU feedback to guide their optimizations.